CLEAN CODE

-ROBERT.C. MARTIN

Writing clean code is like riding a bicycle, you can learn like physics equation in one page but practising it is difficult

<https://github.com/JuanCrg90/Clean-Code-Notes>

If you say "I will back to fix it later" you could fall in the [LeBlanc's law](https://en.wikipedia.org/wiki/Talk%3AList_of_eponymous_laws" \l "Proposal_to_add_LeBlanc.27s_law) "Later equals never"

What is clean code?

A clean code is code that can be read easily. A clean code is code that is taken care of

boy Scout rule:

The code should be taken care of over time.

Always leave the campground cleaner than you found it.

//wrong way

public static void copyChars(char a1[], char a2[]) {

for (int i = 0; i < a1.length; i++) {

a2[i] = a1[i];

}

}

//right way

public static void copyChars(char source[], char destination[]) {

for (int i = 0; i < source.length; i++) {

destination[i] = source[i];

}

}

Appendix-1

void process(final Socket socket) {

if (socket == null )

return;

Runnable runner = new Runnable ( {

public void run() {

try {

String message = MessageUtils.getMessage(socket);

MessageUtils.sendMessage(socket, "Processed: " + message);

closeIgnoringException(socket);

} catch (Exception e) {

e.printStackTrace();

}

} })

Thread th = new Thread(runner);

th.start();

}

SOLID DESIGN PATTERNS:

S = single responsibility principle

O = Open closed principle

L = Liskov substitution principle

I = Interface segregation principle

D = Dependency inversion principle

Single responsibility principle:

A class should have one, and only one reason to change.

Eg.: hibernate, jpa

hibernate acts as orm(object relatonal mapping) between object and relatonal

database. This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes.

Open closed principle:

Class is opened for extension and closed for modification

Liskov substitution principle:

“[subclasses](https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf)should be substitutable for their base classes“, meaning that code expecting a certain class to be used should work if passed any of this class’ subclasses.

Interface segregation principle:

 larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.

Dependency inversion principle:

**public** **class** **Windows98Machine** {

**private** **final** StandardKeyboard keyboard;

**private** **final** Monitor monitor;

**public** **Windows98Machine**() {

monitor = **new** **Monitor**();

keyboard = **new** **StandardKeyboard**();

}

}

**public** **interface** **Keyboard** { }

**public** **class** **Windows98Machine**{

**private** **final** Keyboard keyboard;

**private** **final** Monitor monitor;

**public** **Windows98Machine**(Keyboard keyboard, Monitor monitor) {

this.keyboard = keyboard;

this.monitor = monitor;

}

}

**public** **class** **StandardKeyboard** **implements** **Keyboard** { }

Refer:

<https://github.com/eugenp/tutorials/tree/maGetster/patterns>